PROGRAMMATION EN JAVA

CHAPITRE 1 — Les types de données en Java

2.1 — Les types primitifs

Java possède 8 types primitifs : `byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`.

Les plus utilisés :

🔹 int

Nombre entier sur 32 bits.

```
int age = 25;
```

🔹 double

Nombre à virgule double précision.

```
double prix = 19.99;
```

🔹 float

Nombre à virgule simple précision (nécessite un **f** à la fin).

```
float temperature = 36.5f;
```

🔹 boolean

Valeur logique.

```
boolean estConnecte = true;
```

🔹 char

Un seul caractère (toujours entre **apostrophes** `' '`).

```
char initiale = 'A';
char symbole = '#';
```

2.2 — Les chaînes de caractères : `String`

🔹 Différence entre `char` et `String`

Type	Contenu	Exemple	Guillemets
char	un seul caractère	`'a'`	apostrophes `' '`
String	une suite de caractères (mot, phrase)	`"bonjour"`	guillemets `" "`

Exemple :

```
char lettre = 'J';
String mot = "Java";
```

🔹 Manipulations courantes avec String

```
String nom = "Alice";
System.out.println(nom.length());      // longueur
System.out.println(nom.toUpperCase()); // ALICE
System.out.println(nom.charAt(0));     // A
```

2.3 — Casting (conversion de type)

🔹 De petit vers grand (automatique)

```
int x = 10;
double y = x;   // ok, conversion automatique
```

🔹 De grand vers petit (obligatoire)

```
double a = 9.7;
int b = (int) a;   // b vaut 9
```

2.4 — Constantes

Avec le mot-clé **final** :

```
final double TAUX = 1.08;
```

```
Parfait, on continue !
```

CHAPITRE 2 — Les structures de contrôle en Java

2.1 — Les conditions

🔹 if / else if / else

Permet d'exécuter du code selon une condition.

```
int age = 20;

if (age >= 18) {
    System.out.println("Majeur");
} else if (age >= 13) {
    System.out.println("Adolescent");
} else {
    System.out.println("Enfant");
}
```

2.2 — L’opérateur ternaire

Version courte d’un `if / else`.

```
int note = 15;
String resultat = (note >= 10) ? "Réussi" : "Échoué";

System.out.println(resultat);
```

Structure générale :

```
(condition) ? valeurSiVrai : valeurSiFaux
```

2.3 — Le switch

Permet de tester plusieurs valeurs d’une même variable.
En Java moderne, on peut utiliser **switch avec flèches**.

🔹 Version classique

```
int jour = 3;

switch (jour) {
    case 1:
        System.out.println("Lundi");
        break;
    case 2:
        System.out.println("Mardi");
        break;
    case 3:
        System.out.println("Mercredi");
        break;
    default:
        System.out.println("Autre jour");
}
```

🔹 Version moderne (Java 14+)

```
String saison = switch (3) {
    case 1 -> "Hiver";
    case 2 -> "Printemps";
    case 3 -> "Été";
    case 4 -> "Automne";
    default -> "Inconnu";
};

System.out.println(saison);
```

2.4 — Les boucles

🔹 for

Quand on connaît le nombre d’itérations.

```
for (int i = 0; i < 5; i++) {
    System.out.println("i = " + i);
}
```

🔹 while

S’exécute tant que la condition est vraie.

```
int x = 0;

while (x < 3) {
    System.out.println("x = " + x);
    x++;
}
```

🔹 do…while

Effectue **au moins une fois** le bloc, même si la condition est fausse.

```
int n = 5;

do {
    System.out.println("n = " + n);
    n--;
} while (n > 0);
```

🔹 foreach (boucle améliorée)

Utilisé pour parcourir un tableau ou une collection.

```
String[] fruits = {"Pomme", "Banane", "Orange"};

for (String f : fruits) {
    System.out.println(f);
}
```

2.5 — Les mots-clés break et continue

🔹 break

Quitte une boucle immédiatement.

```
for (int i = 0; i < 10; i++) {
    if (i == 5) break;
    System.out.println(i);
}
```

🔹 continue

Saute une itération.

```
for (int i = 0; i < 5; i++) {
    if (i == 2) continue;
    System.out.println(i);
}
```

Chapitre 2 — Résumé rapide

- **if, else if, else** pour les tests classiques
- **ternaire** pour une condition courte
- **switch** pour tester plusieurs valeurs
- **for, while, do while, foreach** pour les répétitions
- **break / continue** pour contrôler les boucles

CHAPITRE 3 — Les méthodes (fonctions) en Java

3.1 — Qu'est-ce qu'une méthode ?

Une **méthode** (ou fonction) est un bloc de code réutilisable.
Elle peut :

- exécuter une action
- recevoir des paramètres
- retourner une valeur
- être `static` ou non

Structure générale :

```
modificateur typeRetour nomMéthode(paramètres) {
    // code
}
```

3.2 — Méthode simple (sans paramètres)

```
public static void direBonjour() {
    System.out.println("Bonjour !");
}
```

- **Comment appeler cette méthode ?**

```
direBonjour();
```

3.3 — Méthode avec paramètres

```
public static void direBonjourA(String nom) {
    System.out.println("Bonjour " + nom + " !");
}
```

- **Appel**

```
direBonjourA("Alice");
```

3.4 — Méthode qui retourne une valeur

```
public static int addition(int a, int b) {
    return a + b;
}
```

- **Appel**

```
int resultat = addition(5, 3);
System.out.println(resultat);  // 8
```

3.5 — Méthodes `static` vs méthodes d’instance

- **Méthode static**

Appelée sans créer d’objet.

```
public static void test() {}
test();  // OK
```

🔹 Méthode d’instance

Nécessite un **objet**.

```
class Personne {
    public void parler() {
        System.out.println("Je parle !");
    }
}

// Appel :
Personne p = new Personne();
p.parler();
```

`Une méthode static appartient à la *classe*`

C’est-à-dire que **le code n’a pas besoin d’un objet pour exister**.

Exemples typiques :

- `Math.sqrt(9)`
- `main()` est static
- fonctions utilitaires

`Quand utiliser non-static ?`

✔️ `La méthode utilise des attributs de l’objet`

✔️ `La méthode modifie l’état d’un objet`

✔️ `La méthode dépend du contexte (ex : un utilisateur, un compte, une voiture…)`

Type	Appartient à	On l’appelle avec	Sert pour
static	la classe	NomClasse.methode()	outils, calculs, méthodes communes
non static	un objet	objet.methode()	actions liées à des données d’objet

3.6 — Surcharge de méthode (overloading)

Plusieurs méthodes peuvent avoir **le même nom**, mais **des paramètres différents**.

```
public static int calcul(int a, int b) {
    return a + b;
}

public static double calcul(double a, double b) {
    return a + b;
}
```

3.7 — Paramètres multiples, ordre et types

```
public static void info(String nom, int age) {
    System.out.println(nom + " a " + age + " ans.");
}

info("Alex", 22);
```

3.8 — Retourner rien : type `void`

```
public static void afficher() {
    System.out.println("Rien à retourner");
}
```

3.9 — Bonnes pratiques

- Noms clairs (`calculTotal`, `getNom`…)
- Une méthode = une action
- Limiter le nombre de paramètres (sinon utiliser un objet)

Bonus

Le rôle de `toString()` en Java

`toString()` est une **méthode spéciale** présente dans **toutes les classes Java**, car elle vient de la classe **Object**, la classe mère de tout.

À quoi sert `toString()` ?

Elle sert à **représenter un objet sous forme de texte**.

Exemple :
Quand tu fais :

```
System.out.println(objet);
```

Java fait en réalité :

```
System.out.println(objet.toString());
```

Exemple sans toString() personnalisé

```
class Personne {
    String nom;
    int age;

    Personne(String nom, int age) {
        this.nom = nom;
        this.age = age;
    }
}

Personne p = new Personne("Alice", 20);
System.out.println(p);
```

Résultat par défaut (pas très lisible) :

```
Personne@7a81197d
```

Pourquoi ?
Parce que Java affiche le **nom de la classe + un code interne** → pas utile.

Version avec toString() personnalisé

```
class Personne {
    String nom;
    int age;

    Personne(String nom, int age) {
        this.nom = nom;
        this.age = age;
    }

    @Override
    public String toString() {
        return "Personne(nom=" + nom + ", age=" + age + ")";
    }
}
```

🔹 Utilisation

```
Personne p = new Personne("Alice", 20);
System.out.println(p);
```

Résultat :

```
Personne(nom=Alice, age=20)
```

Beaucoup plus propre !
`toString()` est souvent utilisé pour :

- **afficher les objets dans des logs**
- **débugger**
- **afficher proprement les données**

Résumé du Chapitre 3

Une méthode :

- peut être `static` ou liée à un objet
- peut avoir ou non des paramètres
- peut renvoyer une valeur ou `void`
- se **définit** une seule fois mais peut être **appelée** partout
- **peut être surchargée**
- Rôle de **toString**

CHAPITRE 4 — Programmation Orientée Objet (POO)

4.1 — Classe

Une classe est un **modèle** qui contient :

- des **attributs** (données)
- des **méthodes** (actions)

```
class Personne {
    String nom;
    int age;

    void parler() {
        System.out.println("Je m'appelle " + nom);
    }
}
```

4.2 — Objet

Un objet est une **instance** d'une classe.

```
Personne p = new Personne();
p.nom = "Alice";
p.age = 21;
p.parler();
```

4.3 — Attributs

Variables qui décrivent l'état d'un objet. Chaque objet possède ses propres valeurs.

```
class Voiture {
    String marque;
    int vitesse;
}
```

4.4 — Constructeurs

Permettent d'initialiser un objet.

```
class Personne {
    String nom;
    int age;

    Personne(String nom, int age) {
        this.nom = nom;
        this.age = age;
    }
}
```

Utilisation :

```
Personne p = new Personne("Bob", 22);
```

4.5 — this

Référence l'objet courant.

✔️ Pas obligatoire si les noms sont différents
✔️ Obligatoire si l'attribut et le paramètre ont le même nom

```
class Personne {
    String nom;

    Personne(String nom) {
        this.nom = nom; // obligatoire ici
    }
}
```

4.6 — Encapsulation (private, public, protected)

- **private**

Attributs **cachés**, accessibles uniquement dans la classe.

- **public**

Méthodes accessibles de l'extérieur.

- **protected**

Comme private, mais les classes enfants peuvent y accéder.

Accesseurs (getters)

Lire une donnée privée.

```
public double getSolde() {
    return solde;
}
```

Modificateurs (setters)

Modifier une donnée privée.

```
public void setSolde(double montant) {
    solde = montant;
}
```

Exemple complet

```
class Compte {
    private double solde;     // protégé
    protected String type;    // accessible en héritage

    public double getSolde() { return solde; }
    public void setSolde(double s) { solde = s; }
}
```

4.7 — toString()

Pour afficher un objet proprement.

```
@Override
public String toString() {
    return "Personne(" + nom + ", " + age + ")";
}
```

4.8 — Héritage

Une classe hérite d'une autre.

```
class Animal {
    void parler() { System.out.println("Un bruit..."); }
}

class Chien extends Animal {
    @Override
    void parler() { System.out.println("Wouf !"); }
}
```

4.9 — Polymorphisme

Une même méthode → différents comportements.

```
Animal a = new Chien();
a.parler();  // Wouf !
```

4.10 — Classes abstraites

Ne peuvent pas être instanciées.

```
abstract class Forme {
    abstract double aire();
```

```
}
```

4.11 — Interfaces

Contiennent uniquement les méthodes à implémenter.

```
interface Deplacable {
    void avancer();
}

class Robot implements Deplacable {
    public void avancer() {
        System.out.println("Le robot avance");
    }
}
```

Résumé clair du Chapitre 4

Concept	Rôle
Classe	Modèle
Objet	Instance
Attribut	État
Méthode	Action
Constructeur	Initialisation
Encapsulation	Protection des données
Getter	Lire
Setter	Modifier
private	Caché
protected	Accessible en héritage
public	Accessible partout
Héritage	Réutilisation
Polymorphisme	Méthodes → comportements différents
Classe abstraite	Modèle incomplet
Interface	Contrat de méthodes

CHAPITRE 5 — Héritage (Java)

5.1 — Définition

L'héritage permet à une classe **d'hériter** d'une autre classe.

- Classe parent = **super-classe**
- Classe enfant = **sous-classe**

- Mot-clé : `extends`

5.2 — Objectif

- Éviter de répéter du code
- Réutiliser des méthodes et attributs
- Organiser les classes de façon logique

5.3 — Exemple simple

```
class Animal {
    String nom;

    void manger() {
        System.out.println("Je mange...");
    }
}

class Chien extends Animal {
    void aboyer() {
        System.out.println("Wouf !");
    }
}
```

5.4 — Utilisation

```
Chien c = new Chien();
c.nom = "Rex";    // hérité de Animal
c.manger();       // hérité
c.aboyer();       // propre à Chien
```

5.5 — Le mot-clé super

`super` permet d'appeler le constructeur ou les méthodes du parent.

```
class Animal {
    Animal() {
        System.out.println("Animal créé");
    }
}

class Chien extends Animal {
    Chien() {
        super(); // appelle le constructeur de Animal
        System.out.println("Chien créé");
    }
}
```

Résumé du chapitre

- `extends` = hériter d'une classe
- La sous-classe récupère tout ce qui n'est pas `private`
- `super` = accéder à la super-classe
- L'héritage permet la **réutilisation du code** et une **meilleure organisation**

CHAPITRE 6 — Polymorphisme (Java)

6.1 — Définition

Le polymorphisme permet à **une même méthode** d'avoir **différents comportements** selon l'objet qui l'utilise.

- Même nom de méthode
- Comportement différent selon la classe

6.2 — Exemple simple

```
class Animal {
    void parler() {
        System.out.println("Un bruit...");
    }
}

class Chien extends Animal {
    @Override
    void parler() {
        System.out.println("Wouf !");
    }
}

class Chat extends Animal {
    @Override
    void parler() {
        System.out.println("Miaou !");
    }
}
```

6.3 — Utilisation

```
Animal a1 = new Chien();
Animal a2 = new Chat();

a1.parler();  // Wouf !
a2.parler();  // Miaou !
```

Même méthode `parler()` → comportements différents selon l'objet.

6.4 — Polymorphisme et héritage

- Le polymorphisme **fonctionne grâce à l'héritage**
- Les méthodes **surchargées ou redéfinies** permettent ce comportement

Résumé du Chapitre

- Polymorphisme = **plusieurs comportements pour une même méthode**
- Utilisé avec **héritage et redéfinition (`@Override`)**
- Permet d'écrire du code **plus flexible et réutilisable**

CHAPITRE 7 — Classes Abstraites et Interfaces (Java)

7.1 — Classes abstraites

Définition

- Une classe abstraite est **un modèle incomplet**
- **Ne peut pas être instanciée** directement
- Peut contenir :
    - des méthodes **abstraites** (sans code)
    - des méthodes normales (avec code)

Exemple

```
abstract class Forme {
    abstract double aire();  // méthode abstraite
    void afficher() {        // méthode normale
        System.out.println("Ceci est une forme");
    }
}
```

Utilisation

```
class Cercle extends Forme {
    double rayon;

    Cercle(double r) {
        rayon = r;
    }

    @Override
    double aire() {
```

```
        return 3.14 * rayon * rayon;
    }
}

Cercle c = new Cercle(5);
System.out.println(c.aire()); // 78.5
c.afficher();                  // Ceci est une forme
```

7.2 — Interfaces

Définition

- Une interface définit **des méthodes à implémenter**, **sans code**
- Une classe peut **implémenter plusieurs interfaces**
- Mot-clé : `interface` et `implements`

Exemple

```
interface Deplacable {
    void avancer();
    void reculer();
}

class Robot implements Deplacable {
    public void avancer() {
        System.out.println("Le robot avance");
    }

    public void reculer() {
        System.out.println("Le robot recule");
    }
}
```

Utilisation

```
Robot r = new Robot();
r.avancer();  // Le robot avance
r.reculer();  // Le robot recule
```

✅ Les interfaces permettent de **forcer certaines méthodes** dans les classes qui les implémentent.

7.3 — Résumé rapide

Concept	Rôle
Classe abstraite	Modèle incomplet, peut contenir du code et des méthodes abstraites
Méthode abstraite	Doit être redéfinie dans la sous-classe
Interface	Contrat sans code, impose l'implémentation des méthodes

Concept	Rôle
implements	Permet à une classe d’utiliser une interface
@Override	Redéfinition d’une méthode (classe abstraite ou interface)

CHAPITRE 8 — Surcharge et Redéfinition de Méthodes (Java)

8.1 — Surcharge de méthodes (Overloading)

Définition

- Même **nom de méthode**, mais **paramètres différents**
- Permet de **réutiliser un nom logique** pour plusieurs variantes

Exemple

```
class Calcul {
    int addition(int a, int b) {
        return a + b;
    }

    double addition(double a, double b) {
        return a + b;
    }

    int addition(int a, int b, int c) {
        return a + b + c;
    }
}
```

Utilisation

```
Calcul calc = new Calcul();
System.out.println(calc.addition(2,3));       // 5
System.out.println(calc.addition(2.5,3.5));   // 6.0
System.out.println(calc.addition(1,2,3));     // 6
```

✅ Les **types et le nombre de paramètres** permettent de distinguer les méthodes.

8.2 — Redéfinition de méthodes (Overriding)

Définition

- Une **sous-classe** redéfinit une méthode **héritée de sa super-classe**
- Même nom, même type de retour, mêmes paramètres
- Sert à **modifier le comportement hérité**

Exemple

```
class Animal {
    void parler() {
        System.out.println("Un bruit...");
    }
}

class Chien extends Animal {
    @Override
    void parler() {
        System.out.println("Wouf !");
    }
}
```

Utilisation

```
Animal a = new Chien();
a.parler();  // Wouf !
```

✅ `@Override` est **optionnel mais recommandé** pour éviter les erreurs.

8.3 — Différence rapide

Concept	Quand l'utiliser	Exemple
Surcharge (Overload)	Même méthode, paramètres différents	addition(int a, int b) / addition(double a, double b)
Redéfinition (Override)	Méthode héritée d'une super-classe	parler() dans Chien hérite d'Animal

CHAPITRE 9 — Tableaux et Collections (Java)

9.1 — Les tableaux (arrays)

Définition

- Un tableau est une **structure qui stocke plusieurs valeurs du même type**
- Taille **fixe** après création

Déclaration et initialisation

```
// Déclaration + création
int[] notes = new int[5];
```

```
// Initialisation
notes[0] = 10;
notes[1] = 12;
notes[2] = 15;
notes[3] = 18;
notes[4] = 20;

// Déclaration + initialisation en même temps
String[] fruits = {"Pomme", "Banane", "Orange"};
```

Parcourir un tableau

```
// Boucle classique
for (int i = 0; i < fruits.length; i++) {
    System.out.println(fruits[i]);
}

// Boucle foreach
for (String f : fruits) {
    System.out.println(f);
}
```

9.2 — Tableaux multidimensionnels

- Un tableau peut contenir **des tableaux** → utile pour matrices

```
int[][] matrice = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

// Parcours
for (int i = 0; i < matrice.length; i++) {
    for (int j = 0; j < matrice[i].length; j++) {
        System.out.print(matrice[i][j] + " ");
    }
    System.out.println();
}
```

9.3 — Collections (ArrayList, HashMap, etc.)

9.3.1 — ArrayList

- Tableau **dynamique** → peut changer de taille
- Import : `import java.util.ArrayList;`

```
ArrayList<String> liste = new ArrayList<>();
liste.add("Alice");
liste.add("Bob");
liste.add("Charlie");
```

```
System.out.println(liste.get(0)); // Alice
liste.remove(1); // supprime Bob

for (String nom : liste) {
    System.out.println(nom);
}
```

9.3.2 — HashMap

- Stocke des **paires clé-valeur**
- Import : `import java.util.HashMap;`

```
HashMap<String, Integer> notes = new HashMap<>();
notes.put("Alice", 15);
notes.put("Bob", 12);

System.out.println(notes.get("Alice")); // 15

for (String key : notes.keySet()) {
    System.out.println(key + " : " + notes.get(key));
}
```

9.3.3 — Autres collections utiles

Collection	Caractéristiques
LinkedList	Liste chaînée, insertion/suppression rapide
HashSet	Ensemble, pas de doublons
TreeSet	Ensemble trié
HashMap / TreeMap	Stockage clé-valeur, tri possible (TreeMap)

9.4 — Différence tableau / ArrayList

Tableau	ArrayList
Taille fixe	Taille dynamique
Type primitif possible	Doit utiliser objet (Integer, Double…)
Syntaxe simple	Méthodes plus puissantes (`add`, `remove`, `contains`)

9.5 — Résumé

- **Tableaux** : structure simple et fixe
- **Tableaux multidimensionnels** : pour matrices
- **ArrayList** : tableau dynamique, très utilisé
- **HashMap** : stockage clé-valeur

- Collections permettent de manipuler les données **plus facilement et efficacement**

CHAPITRE 10 — Exceptions et Gestion des Erreurs (Java)

10.1 — Définition

- Une **exception** est une **erreur détectée pendant l'exécution** du programme
- Permet de **gérer les erreurs sans faire planter le programme**
- Exemples : division par zéro, tableau hors limites, conversion invalide…

10.2 — Les types d'exceptions

Type	Description	Exemple
Checked (vérifiée)	Doit être gérée ou déclarée	`IOException`
Unchecked (non vérifiée)	Erreurs à l'exécution, pas obligé de gérer	`ArithmeticException`, `NullPointerException`
Error	Problèmes graves, rarement gérés	`OutOfMemoryError`

10.3 — Structure try / catch / finally

```
try {
    int resultat = 10 / 0;  // peut lancer ArithmeticException
    System.out.println(resultat);
} catch (ArithmeticException e) {
    System.out.println("Erreur : division par zéro !");
} finally {
    System.out.println("Bloc finally exécuté toujours");
}
```

- **try** : code qui peut provoquer une exception
- **catch** : gestion de l'exception
- **finally** : bloc exécuté **toujours**, même si exception ou retour

10.4 — Gérer plusieurs exceptions

```
try {
    int[] tab = new int[3];
    tab[5] = 10; // ArrayIndexOutOfBoundsException
    int x = 10 / 0; // ArithmeticException
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Erreur : indice hors limites");
} catch (ArithmeticException e) {
```

```
    System.out.println("Erreur : division par zéro");
}
```

On peut **chaîner plusieurs catch** pour différents types d'erreurs.

10.5 — Throw et throws

throw

- Sert à **lancer une exception explicitement**

```
void verifierAge(int age) {
    if (age < 18) {
        throw new IllegalArgumentException("Age trop jeune !");
    }
}
```

throws

- Déclare qu'une méthode **peut lancer une exception**

```
void lireFichier(String nomFichier) throws IOException {
    FileReader fr = new FileReader(nomFichier);
}
```

10.6 — Exemples d'exceptions courantes

Exception	Quand elle se produit
`ArithmeticException`	Division par zéro
`ArrayIndexOutOfBoundsException`	Indice de tableau invalide
`NullPointerException`	Objet null utilisé
`NumberFormatException`	Conversion String → nombre invalide
`IOException`	Erreurs liées aux fichiers

10.7 — Bonnes pratiques

- Toujours gérer les exceptions **possibles**
- Ne pas utiliser catch général `Exception` à tout bout de champ
- Utiliser finally pour **fermer fichiers / ressources**
- Créer ses propres exceptions si nécessaire

10.8 — Résumé

- `try` = exécuter le code à risque
- `catch` = gérer l'erreur

- `finally` = s’exécute toujours
- `throw` = lancer une exception
- `throws` = déclarer qu’une méthode peut lancer une exception
- Exceptions = **gestion d’erreurs propre et sûre**